Prodo

Language Specifications

Cañete, Jamyl Marie
Fiestada, Vincent Paul
Infante, Patricia
Joaquin, Neil Jonathan
Mase, James Reynald

For the most recently updated version, go to http://bit.ly/prodolang_specs
There is a Python implementation of Prodo on GitHub: https://github.com/prodo-team/prodo

Contents

Overview

These are the functional specifications for *Prodo*, an imperative programming language. The word "prodo" can be translated as "assign" in Latin. Having decided to create a language around the imperative programming paradigm, the creators agreed to associate the name of the language to a word related to the chosen paradigm; since the imperative language focuses on performing tasks through assignment of variables, the writers settled on the word 'assign'. It was also decided that the name of the language should be simple, memorable, and instantly appealing.

Imperative Paradigm

Prodo supports the imperative programming paradigm, which is closest to the actual structure of computers. It is a model based on moving bits and changing machine state. An imperative program tells the machine how to do something by way of concise instructions.

Structure

The basic structure of a program *Prodo* shall be a sequence of statements that are executed in succession. The sequential flow of execution can be modified by conditional and looping statements. Variables play a key role as they serve as abstractions of hardware in memory cells. In imperative programming, changing the values of variables is analogous to changing machine state towards the goal of the program. Thus, the assignment statement is a very important and frequently used statement.

Inspiration

The authors were inspired by the imperative programming languages they are familiar with such as C, Java, and Python. Being exposed to these languages for a long time would prove helpful in creating *Prodo*. Furthermore, having researched on the other paradigms, e.g. functional, object-oriented and logic, the authors decided that imperative is the most intuitive, flexible, and easy to learn.

Imperative programming is arguably the most common paradigm in use. It seems to mirror the most common and immediate way that humans think of solutions to problems--in terms of steps or instructions. Thus it is much easier to grasp and arguably more flexible than declarative paradigms such as functional and logic programming. In creating *Prodo*, the goal is to develop a language and give it the best features in the writers' favorite languages.

Lexical and Syntax Analysis

The current implementation of *Prodo* is a Python parser coupled with simple and lightweight debugging. The Python parser is an LL(1) parser generated by a dependency of the implementation: YAPPS, freely available on GitHub: https://github.com/smurfix/yapps.

The team has written a simple EBNF-like grammar for *Prodo*, expressed as a combination of Python code for the semantic side and regular expressions for the syntactic side. YAPPS is used to generate an LR parser from this EBNF-like grammar. The LL(1) parser generated by YAPPS runs on Python and is able to translate *Prodo* code into Python code.

The Python code is then compiled into a code object and executed using the "`compile`" and "`exec`" functions in Python. The translated *Prodo* code also has access to a pre-made *Prodo* library of "built-in" subprograms.

Please note that the translation **does not** have to be from *Prodo* to Python. It can be from Prodo to C++, to C, or even to Machine Code. However, for the sake of simplicity, ease of implementation, and the constraints of the project, Python has been chosen as an implementing and runtime language.

By convention, *Prodo* code files should have the file extension *.prodo.

Names

Identifiers or names are used to refer to functions, variables & constants, instructions, etc. This document will use the two terms interchangeably. In deciding the form of identifiers in *Prodo*, the authors looked at popular languages and sought a form that is readable, looks good (not visually jarring) and flexible.

Case Sensitivity

Names in *Prodo* shall be **case-sensitive**, i.e. two or more names with different cases of similar characters are still distinct from each other. Case-sensitivity was favored because it was already a common practice in modern programming language. Case-sensitive names improve readability in the sense that most programmers are already used to it. Case-sensitivity also puts the jarring look of all caps out of the question. Finally, case-sensitivity allows for a larger pool of possible names.

Name Form

A name in *Prodo* shall be a string of characters. The first character is always either a letter of arbitrary case. This may be followed by a string of letters, numbers, or the special characters '`$`' (Dollar Sign), '`@`' (At-sign) and '`_`' (Underscore) in any order. A name must be at least 1 character long. The purpose of allowing the '`$`', '`@`', and '`_`' symbols are to provide a facility for programmers' coding conventions. An underscore can be used in lieu of spacing. A '`$`' or '`@`' may be used for specific conventions that will not be determined here. In any case, by allowing these characters, names can be more flexible. The writers have also remarked that restricting the first character to be a letter improves readability and makes differentiating between names and numeric literals easier. The '_' is not allowed as a first character because of its appearance (it may be missed by a glance). Other special symbols as a first character also don't make much sense because humans usually think of names in terms of letters.

Grammar Definition

The formal grammar for an identifier or name in *Prodo* follows:

```
token ID: (a-z | A-Z )( a-z | A-Z | 0-9 | $ | _ | @ )*
```

Lines and Blocks

Code blocks in *Prodo* shall be represented as a series of statements, **each in its own line**. In effect, the line delimiter in *Prodo* is the newline "`\n`" character. This is opposed to some popular programming languages, most notably in the C family (including C++, Java, and C#) where lines are separated by a semicolon "`;`". *Prodo* shall favor the new line because it forces programmers to create more readable code. It prevents one-liners, which if one takes the examples of Python and Ruby, make for greater writability and readability.

Some statements will have to be blocked for code structuring. For instance, conditional structures, iteration structures, and subprogram definitions require code blocks. To group a series of statements into a single block, they **must be grouped contiguously and end with a block ender.** A *block ender* is language element which semantically ends a structured group of code.

Blocks can be ended in three ways:

1) <u>End of File</u> - the file ends. This is only valid for the global or uppermost block of code.
2) <u>'end'</u> - The reserved word **end** is used to end a block of code. This is only valid for blocks other than the global or uppermost block. After the **end** word, a contiguous line of code, i.e. the block, ends. This is used for subprogram definitions, looping statements, and to end the last block of a selection statement.
3) <u>'elseif'</u> and <u>'else'</u> - Selection statements can contain multiple blocks of code. In this case, only one **`'end'`** symbol is expected, at the end of the last block. Preceding blocks end when an **`'elseif'`** or **`'else'`** is encountered.

By convention, lines of code grouped inside a block are indented using four (4) spaces or a tab. However, these indentations are ignored by the parser. The actual grouping lies in the three symbols or flags above.

Lines are read from top to bottom, left to write.

Comments

Any line (or prefixed line) that begins with a tilde (~) is considered as a comment and shall be ignored by the scanner. The comment ends when the line ends, i.e. after the newline character. Multi-line comments are not allowed in *Prodo* to avoid errors in which important code has been commented out. The lack of multi-line comments is also a measure taken to encourage programmers to have short, concise comments in multiple areas of the code.

Some examples of comments:

```
~ This is a comment on its own line
z := 6 ~ A comment prefixed by code
```

Reserved Words and Keywords

What follows is a table of reserved words and their corresponding meanings. (As of the time of this writing, *Prodo* is still a work in progress and is at the early stages of construction. These reserved words may be modified or added to in the future.) Each *reserved word* shall be treated by the *Prodo* parser as a single terminal symbol. They cannot be used for user-defined variables and functions. Primitive type names are not included below.

Reserved word	Meaning
yes	Boolean value: true
no	Boolean value: false
nil	Special Literal Constant, equivalent to Null or None in other languages

for	The keywords 'for', 'to' and 'by' are used in a for loop like this:
	
	`for |i := 1 to 10 by 1|`
	     ...
	
	`for` - define a for loop and begin its "header"
	`to` - precedes the value at which the for loop stops when the iterator `i` is equal to it
	`by` - precedes the value used to increment the iterator `i`
	
	The "by" part can be any integer or real number not equal to zero. It can also be omitted, in which case the default of 1 as incrementor will be used.
	
	In C, the above code would look like:
	`for (i = 1; i != 10; i += 1 )`

to	
by	
while	Indicates a while loop and must be followed by a boolean expression
loop	Indicates a logically controlled while loop with a post-test (or do-while loop)
and	Logical operator AND, between two boolean expressions
or	Logical operator OR, between two boolean expressions

xor	Logical operator Exclusive OR, between two boolean expressions
not	Logical operator NOT, precedes a boolean expression
if	defines a conditional statement, precedes a boolean expression
elseif	defines a second (or third, fourth, & so on…) condition in a conditional (if) statement; must be preceded by an if-block
else	defines the last condition in a conditional (if) statement; must be preceded by an if-block or elseif-block
next	Indicates an instruction to stop executing the current iteration of a loop and immediately move on to the next iteration; must be inside a loop (for/while) block
stop	Indicates an instruction to stop executing the current iteration of a loop and immediately exit the loop; must be inside a loop block
fcn	Defines the following name to be a function or method.
conclude	Immediately stops execution of the current function and returns the value of the expressions after it.
end	Used to end a code block (not indented)

These reserved words are pretty standard. Some of them have been changed with the goal of making their meanings easier to deduce. For instance, instead of ‘return’, the word ‘conclude’ is used because it also implies a stop in execution.

Binding Type

Type Binding in *Prodo* programs shall be **static**, that is, variables remain bound to a single type during its entire lifetime. Type binding shall occur at interpretation time, and the type of a variable is explicitly stated in the program code. Data types will be covered in succeeding sections.

The reason for the team’s decision to use **static type binding** is because it is easier to implement. The interpreter will not need to “guess” the type of the variable at initialization and the complexity involved in implicitly allowing variables to change type is removed. Also, static type binding should make debugging and maintenance of *Prodo* programs easier.

Lifetime and Scoping

All variables in *Prodo* have **static scoping,** hence the visibility of variables will depend solely on the hierarchical position of the code block in which they are declared. This decision was made because dynamic scoping is confusing, at least for the writers.

Lifetime for all variables in *Prodo*, at the time of this writing shall be **stack dynamic**. Hence, a variable instance’s lifetime begins when the program execution enters its scope. Heap-dynamic variables are only useful if the language supports structures or classes, but *Prodo* doesn't at this time. Meanwhile, static lifetime variables can be achieved with stack dynamic variables declared globally (or in the highest order code block). So to simplify things, stack dynamic shall be used.

Data Types

As mentioned above, type binding in *Prodo* shall be static. The data type of a variable must be explicitly stated when it is first declared/initialized in a scope. *Prodo* will support some basic or **primitive data types** that will always be available to any program.

Primitives

The following is a table of primitive data types, their names in the language, possible values, and example use cases for them. These types reflect the primitives also available in most modern programming languages. Type names of Primitives are **reserved words** and may not be overridden.

Primitive Type	Type name	Possible Values	Possible Uses
None	void	`‘nil’`	
Boolean	bool	`‘yes’, ‘no’, ‘0’, ‘1’`	For boolean or logical operations; to denote values that can only be either true or false; as flags in a program
Integer	int	Integer values. The value range shall depend on the machine, but the range of negative values is equal to that of the positive values	Arithmetic operations; loop counters; representation of real-world numbers
Floating Point	real	Signed real or floating-point numbers. Same range of negative and positive values.	Physics computations; representing ratios, fractions, etc. Representing real-world constants, i.e. pi, e
String	str	String of characters, for example `“John Green”`. Contains 0 or more ASCII characters inside double quotes. A ‘char’ may be represented as an ‘str’ with a single character in it.	Input and output operations; string processing

Type Names: Grammar Definition

Data Type Names are identifiers used to identify the type of a variable, constant or return type of functions. The grammar definition is as follows, in BNF:

```
token TYPE: r'[a-zA-Z]'
rule type_name: 'void' | 'bool' | 'int' | 'real' | 'str' | 'array' |
                    | TYPE
```

It is similar to the grammar of identifiers/names, except they can only contain letters. The primitive type names are also hardcoded into the language grammar definition because they are considered as reserved words.

Strings

As indicated in the previous table, strings will be supported in *Prodo* as a primitive data type. Letting them be a primitive data type will improve not only writability of the code but will also be less costly than creating them with arrays.It will help the programmer do simple operations such as concatenation and assignment easier because there is no need for doing loops, unlike when they are made of array of characters.

Basic string operations such as concatenation and getting the length shall be provided for.

Other Types

Other built-in types in *Prodo* are outlined in the following table:

Data Type	Type name	Possible Values	Possible Uses
Array	array	Any number of values of any type, indexed by integers beginning with 0. Example: `array inventory { 1, 67, 32.8, “Panda” }`	Storing a homogenous or heterogenous list of data, delimited by curly braces, each item separated by a comma. The curly braces are a natural choice because they are used in sets in Mathematics.

Record	structure	A fixed number of values of any type, indexed by names; each name is considered an identifier bound to an entry in the record. Each entry is accessed with the dot (.) operator. Example:
		
		`structure person1`
		`    int id := 69`
		`    str name := “Ice King”`
		`    str address := “145”`
		`    real grade := 2.0`
		`end`
			Representing data with various components.

User-Defined Types

Prodo will **not** support user-defined data types. As an imperative language, user-defined data types will have a very minimal influence in making *Prodo* a better language. The best this will do is make *Prodo* more readable while not doing anything to improve functionality. Simply stated, any user-defined type short of classes have minimal benefits to a language. Since *Prodo* is imperative, it does not make sense to support classes.

Instead, *Prodo* will take a tip from Javascript. While users cannot create their own types, they will be able to create structs (essentially fixed-size records) to “combine” existing types into more complex structures. This maintains the simplicity of *Prodo* while maintaining its statically typed nature.

Type Checking

Prodo will be doing the type-checking during compile time as it will help the programmer detect the error earlier. Using **Static Type Checking** reduces runtime overhead and also greatly increases chances of detecting errors. Coupled with **static typing** of variables in *Prodo,* this should greatly improve reliability of code.

Additionally, *Prodo* will allow **explicit type casting** but **not coercion** since it will force the programmer to explicitly cast the desired variables thus giving the programmer more control of the code. Also, it will help the programmer avoid having gibberish outputs that may happen when automatic conversions (coercion) were done. Implicit type conversion reduces code reliability and increases risk of type-related errors. Explicit type conversion makes all type conversions clearly visible.

Expressions and Assignments

Arithmetic Expressions

Arithmetic expressions in *Prodo* programs are written in infix notation, with the following specific conventions: Binary operators appear in between their operands; Increment/Decrement operators (unary) appear after their single operand. Other unary operators appear before their operand. There are no ternary operators. **Infix notation** was chosen because it is more intuitive for people. Most modern programming languages use infix notation because prefix notation is uncommon to the eyes of people, especially mathematicians. Furthermore, converting infix to prefix notation, which can be worked on by a computer more efficiently, is relatively easy.

The following table shows the arithmetic operators in *Prodo,* along with the corresponding symbol, number of operands, and the result the operator produces. **Take note that coercion is not allowed in *Prodo*.** All binary operators therefore only accept operands of the same type, except for String-int Multiplication which accepts a string and an int. So one cannot simply divide an int by a float, or vice versa. One of the operands must first be explicitly cast to the type of the other.

Operator (Symbol)	Number of Operands	Example	Result
Addition/Concatenation (‘+’)	Binary	`2 + a`	Sum of the two operands (int & real); Concatenated string (str)
Subtraction (‘-’)	Binary	`2 - a`	Difference of the two operands, where the second one is the subtrahend. (int & real)
Multiplication (‘*’)	Binary	`2 * a`	Product of the two operands (int & real)
String-Int Multiplication (‘*’)	Binary (str and int)	`“Nerdfighter” * 6`	Concatenation of *n* instances of the string, where *n* is the int operand
Division (‘/’)	Binary	`2 / a`	Quotient of the two operands, where the second one is the divisor (int & real)

Modulus (‘`%`’)	Binary	`2 % a`	Remainder of the division involving the two operands (int & real)
Negation (‘`-`’)	Unary	`-a`	Opposite-signed version of the operand (int & real)
Increment (‘`++`’)	Unary	`a++`	The operand is assigned the value 1 + its previous value (int & real)
Decrement (‘`--`’)	Unary	`a--`	The operand is assigned the value of its previous value - 1 (int & real)
Groupings ‘`[`‘ and ‘`]`’	n/a	`a+[a - b]`	Gives the expression inside the grouping pair a higher evaluation precedence. The most deeply nested grouping(s) are evaluated first

A Note on Unary Incrementors

Unary incrementors (++ and --) can only be applied as a single, unaccompanied line. They also do not return a value the same way that binary operators do. This is because they behave the same way as assignment operators do, and have a “side effect”. To avoid confusing the programmer and to simplify code, they **must be on their own lines**.

Operator Overloading

Operators cannot be overloaded by the user. Operations concerning user-defined data types should be dealt with using functions, because operator overloading can lead to diminished readability if the user is not careful.

However, **some primitive (built-in) data types have overloaded operators**. In particular, Addition, Subtraction, Multiplication, Division, Modulo, Negation, Increment and Decrement, are overloaded for the ‘int’ and the ‘real’ data type. Addition and Multiplication operators are also overloaded for strings. The addition operator can involve two str operands (meaning concatenation). The multiplication operator can involve an str and an int operand (see above table).

Type Conversions

It’s been mentioned in the previous sections that *Prodo* shall **not allow coercion**. All type conversions **must be in the form of explicit casting** by the programmer or user. Narrowing will happen when a **real is cast to an int**, when the fractional part will be dropped. Widening will happen when an **int is cast as a real,** because no data will be lost. The fractional part will be 0.0.

Example: `int(3.14)`

Relational and Boolean Expressions

The following are table contains the relational operators in *Prodo* and some relevant information about them. All relational operators yield boolean values.

Operator (Symbol)	Number of Operands	Example	Result
Equal (‘==’)	Binary	`2==a`	True if the two operands are equal (both type and value must be equal); False otherwise (all types)
Greater Than (‘>’)	Binary	`2 > a`	True if the first operand is greater than the second (int, real, str)
Less Than (‘<’)	Binary	`2 < a`	True if the first operand is less than the second (int, real, str)
Greater Than or Equal To (‘>=’)	Binary	`2 >= a`	True if first operand is greater than or equal to the second
Less Than or Equal To (‘<=’)	Binary	`2 <= a`	True if the first operand is less than or equal to the second
Not Equal (‘`!=`’)	Binary	`2 != a`	True if the two operands are not equal (all types)

Boolean Expressions

Boolean expressions are those that yield a ‘bool’ value. ‘bool’ is a primitive data type in *Prodo*. The following table shows the supported boolean operators. Boolean operators accept boolean expressions as operands and yield boolean values.

Operator (Symbol)	Number of Operands	Example	Meaning
And (`‘and’`)	Binary	`2==a and b>=3`	Logical AND
Or (`‘or’`)	Binary	`2==a or b>=3`	Logical OR
Exclusive Or (`‘xor’`)	Binary	`2==a xor b>=3`	Logical XOR
Not (`‘not’`)	Unary	`not 2==b`	Logical NOT

Assignment Operators

The assignment operator for *Prodo* is the symbol ‘`:=`’. Although most programming languages use ‘=’ as an assignment operator, many beginning programmers confuse this with mathematical equivalency. To avoid this confusion, a different symbol was selected, which is already used by mathematicians and logicians.

Other assignment operators, `+=, *=, /=, -=, %=` are also supported. Their meanings are similar to those in languages like C and Javascript. For example:

`a += b` is equivalent to `a := a + b`, and so on for the similar operators. In general, for each ***?*** of the built-in binary arithmetic operators in *Prodo,* an operator ***?=*** is defined such that **a ?= b** is equivalent to **a := a ? b**, where “?” is a binary arithmetic operator.

Simple Assignment

A simple assignment statement involves a left-hand operand (a variable or a variable declaration), the ‘`:=`’ operator, and the value to be assigned as right operand. When declaring a variable for the first time, the **type must be specified** and it **must be initialized** to a value that can be coerced into the type. This is the only time that a value will be coerced, because type checking and casting cannot occur yet for the new variable until it has been initialized.

Example:
```
int x := 5
int y := x
x += y
```

Multiple Assignment

Multiple variables can be in a single assignment statement if and only if:

1. They have the same type
2. They are being assigned the same value
3. The are being declared; i.e., multiple declaration is allowed but multiple assignment to existing variables is not. This is because operators like *=, += and so on can be confusing when used in a multiple assignment statement.
4. The operator **`:=`** is being used

In a multiple variable declaration, the variables are separated by commas.

Example: `int x, y, z := 7`

Grammar Definition for Expressions and Assignment Statements

```
rule exp_statement : declaration_exp
                   | identified_exp
rule identified_exp : identifier
                      ((assignment_op
                        additive_exp
                       )
                        |
                        ("(" list_plain ")")
                        | “++”
                        | “--”
                      )
rule declaration_exp : type_name list_identifiers ':=' additive_exp
rule structure_declaration : 'structure' list_identifiers
                               struct_compound_stat

rule identifier : ID
rule assignment_op : ":="
                   | "*="
                   | "/="
                   | "%="
                   | "+="
                   | "-="
rule list_literal : '{' list_plain '}'
rule list_plain : (additive_exp
                  ( "," additive_exp
                  )*
                  | ''
                  )
rule list_identifiers : identifier
                        ( ',' identifier
                        )*
rule additive_exp: term
                   ( ‘+’ term
                   | ‘-’ term
                   )*
rule cast_exp: type_name ‘(‘ + additive_exp + ‘)’
rule term: factor
           ( ‘*’ factor
           | ‘/’ factor
           | ‘%’ factor
           )*
rule factor : INT
            | REAL
            | list_literal
            | 'nil'
            | STRING
            | ( identifier
                ( ‘(‘ list_plain ‘)’
                | ‘[‘ additive_exp ‘]’
                | ''
                )
              )
            | cast_exp
            | ‘[‘ additive_exp ‘]’
            | '-' additive_exp

rule boolean_exp: logical_exp | boolean_literal
rule boolean_literal : 'yes' | 'no' | '1' | '0'                          rule
logical_exp : relational_exp
                  [(‘and | ‘or | ‘xor’) relational_exp]
rule relational_exp: additive_exp relational_op additive_exp
                   | 'not' relational_exp
rule relational_op: '==' | '!=' | '<' | '>' | '<=' | '>='
rule iterative_statement: 'for' ‘|’ [type_name] identifier
                           ':=' additive_exp 'to' additive_exp
                           [‘by’ additive_exp] ‘|’
                          compound_statement
                        | 'while' '|' boolean_exp '|'
                  compound_statement
                        | 'loop'
                          compound_statement
                           '|' boolean_exp '|'
```

Statement-Level Control Structures

Prodo shall provide the standard types of statement-level control structures that can be found in modern programming languages, i.e., if-elseif-else statements or conditionals and for, while, do while loops or iterative control structures. By and large, these are based on contemporary programming languages.

Selection Statements/Conditionals

The selection statements in *Prodo* can be two-way (if-else) or multiple-way (if-elseif-else). They shall provide a means for the programmer to control the flow of execution based on expressions that evaluate to **boolean** values.

Two-Way Selectors

The reserved word *if* shall be used to indicate a selection structure. It is always followed by the control expression (enclosed inside a pair of pipes '|'). The control expression is a boolean value or an expression that evaluates to a boolean value. Other types can be used as control expressions but **must be cast as booleans**. `nil`, an empty string, `0` or `0.0` can be explicitly cast to the type `bool`, producing a value of `'no'` (false). All other values when cast to bool produce `'yes'` (true). Below is an example of the if statement syntax for *Prodo*. (Note that the 'else' clause is optional)

```
int x, y := 4
if | yes |
    x := 1 + y
else
    y := 1 + x
end
```

The above code snippet, when executed, gives x the value 5 and y, 4. As expected, the if-then block executes if the control expression is true. The else-then block executes otherwise. The control expression is enclosed by a pair of pipes '|' to make parsing easier. The pipes are used instead of the conventional '(' to avoid the confusion between subprogram calls.

Multiple-Way Selectors

Multiple-way selectors follow directly from the definition of two-way selectors. But instead of the else block following immediately after the first if block, one or more (theoretically infinitely many) elseif blocks can be inserted in between. The reserved word `'elseif'` is used:

```
int x,y := 4
if | x == 5 |
    x := 1 + y
elseif | x == 4 |
    x := 2 + y
else
    y := 1 + x
end
```

The code snippet above gives x the value of 6 and y remains equal to 4. The form of the control expression after the ‘elseif’ reserved word is the same as the control expression for ‘if’. Switches are not supported in *Prodo*.

Iterative/Looping Statements

Prodo shall provide three statement-level control structures to facilitate iteration, namely: a counter-controlled iterator (`for-to-by`), a logic-controlled iterator with pretest (`while`) and a logic-controlled iterator with post-test (`loop-while`)

Counter-Controlled

Prodo provides a for-loop which is a looping structure controlled by one iteration variable (either an int or a real variable). The code snippet below shows its syntax:

```
int x, sum := 0
for | x := 1 to 11 by 1 |
      sum += x
end
```

The code snippet above will give `sum` the value of the sum of the integers from 1 to 10. A more thorough explanation of the syntax follows:

1. The reserved word ‘for’ is followed by the conditions for the iteration, enclosed in a pair of ‘|’
2. The variable used as an iterator (in this case *x*) **must be an int or real** variable
3. The snippet `x := 1` initializes the iterator variable *x*
4. The code block will loop **while x is not equal to the value following the reserved word ‘to’**.
5. The value following the reserved word **by** serves as the increment of the iterator *x*. It can be positive or negative.

```
for | x := a to b by c |
     LOOPBODY
end
```

The logic goes like so:

1. assign **a** to **x**
2. Is **x** equal to **b**? If **yes: goto 5**; **no: goto 3**
3. execute **Loop Body**
4. assign **x + c** to **x; goto 2**
5. end loop

Note that the “by” part can be any nonzero integer or real value. It can also be omitted, in which case the default increment of 1 will be used.

Logic-Controlled with Pretest

The syntax of while loops in *Prodo* is very similar to contemporary programming languages. The reserved word ‘while’ is followed by the control expression, which is **must be a boolean** or an expression that evaluates to a boolean. The control expression is enclosed inside a pair of ‘|’ or pipes. An example is shown below:

```
int x := 1
int sum := 0
```

```
while | x < 11 |
    sum += x
    x++
end
```

The code snippet above produces the same result as the previous for loop. The logic is as follows:

1. The **control expression** is evaluated.
2. If it is **yes: goto 3, no: goto 4**
3. Execute the **Loop Body; goto 1**
4. end loop

Logic-Controlled with Posttest

Prodo provides a logic-controlled looping structure, similar to do-while loops in other languages, with a post-test. It is indicated by the reserved word 'loop', followed by a code block, then a the post-test boolean expression. An example follows:

```
int x := 1
int sum := 0
loop
    sum += x
    x++
end | x < 11 |
```

The above code snippet produces the same result as the previous for-loop and while-loop. The logic is outlined below:

1. Execute the **Loop body**
2. The **control expression** is evaluated
3. If **yes: goto 1; no: goto 4**
4. end loop

The only difference with a while-loop is that the control expression is checked after the loop body is executed. As a consequence, the loop body is **guaranteed at least one (1)** iteration.

Unconditional Jumps

Prodo does not have an equivalent for the *goto* statement since nothing it can do cannot be replicated by more readable and reliable control structures. The language *does* provide three unconditional jumping statements, namely **`'next'`**, (equivalent of continue) which stops execution of a loop and moves on to the next iteration; and **`'stop'`** (equivalent of break) which stops execution of a loop and prevents further iterations. Semantically, they must only be found **inside compound statements of iterative structures**. Lastly, there is **`'conclude'`** (equivalent of return) which stops execution of a subprogram and returns control to the caller. It may be followed by an return value.

Nesting

Control structures (selection and iteration) can be nested by adding a control statement inside the code block of another. See section "Lines and Blocks" above for more information.

Grammar Definition for Control Structures, Statements

```
token NEWLINE: ('\n')+
token INT: (0-9)+ | '-'(0-9)+
token REAL: (0-9)+ '.' (0-9)+
          | '-' (0-9)+ '.' (0-9)+
token STRING: '"' (ANY)* '"'
token ID: (a-zA-Z) (a-zA-Z0-9$_@)*
token TYPE: (a-zA-Z)
ignore: ' '
ignore: '~' (ANY)*

rule super: ( statement_upper NEWLINE )*
rule statement_upper: exp_statement
                    | fcn_definition
                    | conditional_statement
                    | iterative_statement
                    | structure_declaration
                    | '~' (ANY)*

rule statement : exp_statement
               | jump_statement
               | conditional_statement
               | iterative_statement
               | structure_declaration
               | '~' (ANY)*

rule compound_statement : NEWLINE (statement NEWLINE)+ 'end'
rule p_compound_statement : NEWLINE (statement NEWLINE)+
rule struct_compound_stat : NEWLINE (declaration_exp NEWLINE)+ 'end'
rule jump_statement : 'conclude'
                         ( additive_exp
                         | boolean_literal
                         | ''
                         )
                    | 'next'
                    | 'stop'

rule conditional_statement: 'if' '|' boolean_exp '|'
                            p_compound_statement
                            (elseif_statement)*
                           (else_statement
                           | 'end'
                           )
rule elseif_statement : 'elseif' '|' boolean_exp '|'
                        p_compound_statement
rule else_statement : 'else'
                      compound_statement
```

Subprograms

Subprograms in *Prodo* has 3 basic characteristics: (a) each subprogram has a single entry point; (b) there is only one subprogram executing at any given time, that is, the caller is suspended during the execution of the subprogram, and; (c) control always returns to the caller when the subprogram execution terminates.

Furthermore, unlike other statements, subprograms **cannot be nested inside another statement block**. This is because subprograms are always evaluated before the rest of the program, and so nesting is illegal to simplify program structure.

Subprogram Definition and Call

A subprogram in *Prodo* should be defined in the following format:

```
fcn <return type> <function name> (<parameter type> <parameter name>, … )
  <subprogram body>
end
```

- **Declaration**

  A subprogram can be declared anywhere in the program code, given it is defined in the format stated above. Functions are not first-class. If they are declared inside a code block, they are still visible outside, i.e. subprograms are always visible globally **except** in subprograms declared before it.

- **Header**

  Prodo's subprogram header serves several purposes. First, it states that the following block of syntax is a subprogram by using the reserved word '`fcn`'. Second, it provides the type of data the subprogram should return and the name for the subprogram. And lastly, it specifies the list of parameters to be used in the subprogram. When a `fcn` statement is executed, it assigns the given name to the given function body.

- **Parameters**

  There are two ways that a subprogram can gain access to the data that is to process: through direct access to nonlocal variables or through parameter passing. Data passed through parameters are accessed through names that are local to the subprogram. **Subprogram call** statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. The binding of subprogram call parameters to subprogram header parameters – is done by position: the first parameter in the call is bound to the first parameter in the header and so forth.

- **Local Variables**

  Subprograms are allowed to define their own variables, thereby defining local referencing environments. Access to them is restricted to the subprogram in which they are defined. Subprograms can also access variables declared in their parent scopes. It should be noted that scoping in *Prodo* is static and is entirely dependent on code structure, so the only variables accessible to a subprogram are 1) local variables, 2) actual parameters, and 3) global variables.

Below are examples of a function declaration/definition and function call in *Prodo:*

```
fcn real area (real w, real h)
    real a := w * h
    conclude a
end

real w := 8.0
real y := area(w, 4.3)
```

The above code snippet gives the variable y the value of 8.0 * 4.3 or 34.4.

Location of Subprogram Definitions

There are no separate subprogram declarations in *Prodo.* Unlike in C and C++, subprogram definitions must come with their declarations.

A subprogram is always visible everywhere **except** for subprograms declared before it.Think of it as the subprograms being put inside a header, in order of appearance, and then that header is prefixed to the rest of the code. (In fact, this is what the parser does).

This means that subprograms are always visible globally wherever they are defined in the code.

Parameter Passing

Because *Prodo* is an imperative programming language and depends on changing program state, arguments or actual parameters are passed by value.

As for the format of the parameter passing, it is as a list, and the types **must match** that of the formal parameters in the function definition header.

Overloading

Subprograms in *Prodo* can be overloaded. The correct subprogram to call is determined based on the number of arguments passed. To overload a subprogram, a programmer simply has to define a new subprogram with the same name but with a different number of parameters.

Generic Subprograms

Generic subprograms **will not be supported** in *Prodo* because they are detrimental to the benefits of a strongly typed language. Generic subprograms expose a program to more runtime errors that are otherwise easily prevented by the strong typing implemented by *Prodo*.

Recursion

Prodo shall allow recursion of subprograms, which can be facilitated by a subprogram making a call to itself or even to its caller (or higher up the call stack) inside its own body.

Grammar Definition for Subprograms

```
rule fcn_name : identifier
rule fcn_definition : 'fcn' type_name fcn_name '(' param_list ')'
                      compound_statement

rule param_list : type_name identifier ( "," type_name identifier )*
                | ''
```

A Final Note

As a final note in these specifications, the authors would like the reader to realize that the *Prodo* language is meant to be simple and easy to learn, readable but expressive, and have a short syntax that takes tips from languages like Javascript, Python, and Ruby--while allowing for type checking, hence greater reliability, as in languages like C and C++.